Massachusetts Institute of Technology
Artificial Intelligence Laboratory

8 June 1978 AI Memo # 480

Dynamic Graphics using Quasi Parallelism

Kenneth M. Kahn and Carl Hewitt

Abstract

Dynamic computer graphics is best represented as several processes operating in parallel. Full parallel processing, however, entails much complex mechanism making it difficult to write simple, intuitive programs for generating computer animation. What is presented in this paper is a simple means of attaining the appearance of parallelism and the ability to program the graphics in a conceptually parallel fashion without the complexity of a more general parallel mechanism. Each entity on the display screen can be independently programmed to move, turn, change size, color or shape and to interact with other entities.

The scheme presented herein begins with the notion of a quantum of time, or *tick*, within which there are no ordering constraints on events. Each entity or actor decides what it must do upon the next tick. Ticks are a powerful means of controlling parallel processes but are usually at too low a conceptual level for user convenience. Higher-level operations built upon the tick mechanism are presented, most notably the ability to instruct any entity or group of entities to gradually change or move at a rate that is itself changeable by the same operation. To illustrate these ideas a simple celestial mechanics simulation is presented. Upon each tick the velocities and positions of the objects are updated by the gravitational and propulsive forces acting upon them.

Ticks are only one product of an object-oriented programming style. For the best control and the most modularity, graphics programming should be object oriented. Each object displayed, and its parts, should be independently programable. Instead of being passive data, objects should be responsible for the changes in their position or appearance. Instead of a global controller, each object should interact with the others.

This is a revision of a paper to be presented at the Siggraph '78 Conference.

One of the authors (Kahn) is currently supported by an IBM fellowship. The research described herein is being conducted at the Artificial Intelligence Laboratory.

CONTENTS

I. Parallel Processing for Dynamic Graphics

Dynamic graphics is concerned with the display of changing images. Typically there are many different entities or aspects of entities changing simultaneously. To reduce the programming complexity we represent each entity and its parts as a module capable of changing its state and appearance and of interacting with other modules. To simplify the control of these objects we make them independent entities and run them in parallel.

Each entity on the display screen can be thought of as a little person who can be asked to move, change appearance, remember and forget information. These little people, or actors, interact with each other to form a community. This metaphor of computation as a society of interacting entities is especially appropriate for dynamic graphics where it usually easy to anthropomorphize the images on a display, whether they be of DNA strands, engine parts, simple geometric shapes, super-sonic transports, or people. The communities can exist at different levels, for example, there may be a community of people while simultaneously there is a community of arms, legs, and heads associated with each person. Object-oriented computer languages such as Smalltalk, Act 1, or Director[1] are ideal for programming in this style. Though regardless of computer language one can conceptualize one's display as a community of active entities.[2]

---

1. Director is an object-oriented language especially designed for animation and artificial intelligence applications. It was designed and implemented by Kahn. [Kahn 1978] All the examples in this paper are working programs in Director.

2. Of course, the *convenience* with which one can program this way varies greatly from language to language. The object-oriented parallel scheme presented, for example, would be very difficult to implement in a general fashion in any language which did not permit the construction and subsequent evaluation of code. The ability to modify planned actions is important, as is the ability to do part of a planned action and plan to the rest latter.

Apparent Parallelism

To animate the changes of many objects simultaneously one needs parallel processing or at least the appearance of having it. In this paper we opt for the latter in the interests of simplicity. During a tick, processes can run in any order, even sequentially, so long as the objects are in the desired consistent state when the frame ends. If the animation is being filmed, recorded frame-by-frame on video, or in the computer's memory for later playback then all that matters is that the display is correct when the frame is recorded, between recordings anything may happen. If animation is being displayed in real time then the time to perform all the actions of a tick should be less than a refresh cycle (typically a thirtieth of a second).

To coordinate and control these processes we introduce the notion of a *tick*, or a quantum of time within which one is unconcerned about the order of events. All the objects have associated with them a variable containing a list of actions to take on the next tick. When an object receives a tick it does all the actions it had planned for that time. In the simplest case, an animation program proceeds by sending a tick to each object on the screen, recording or displaying the current state and repeating. It is the responsibility of each object to respond to each tick. More complexity is introduced when there are several ticks to a frame or when only certain subsets of objects are to run at a certain time.

A Simple Example

Suppose we want to animate a shape to move gradually forward and we already have a primitive, called "Forward" that moves an object forward by causing it to hide and reappear at its new position.[1] We could write the following simple program:

---

1. This example and the next rely upon a computational display entity called a "turtle". Turtles have a state consisting of a position and direction and respond to messages asking them to go forward or to turn. More details can be found in [Papert 1971a], [Papert 1971b] and [Goldstein 1975].

```
REPEAT FOREVER (ASK AN-OBJECT FORWARD SPEED) (ASK SCREEN RECORD)
```

The object called "an-object" will go forward "speed" then the screen is recorded and this is repeated forever. If we wanted two objects to move forward simultaneously then we could write:

```
REPEAT FOREVER (ASK OBJECT1 FORWARD SPEED1)
               (ASK OBJECT2 FORWARD SPEED2)
               (ASK SCREEN RECORD)
```

The need for explicitly using ticks have not yet risen. But suppose we want "Object1" to go forward 300 steps and the other 400 steps. Or we want "Object2" to change its speed after four frames. The program becomes more and more unwieldy. An alternative is to explicitly use ticks as follows (as opposed to the implicit use of ticks in the previous examples)[1]

```
(ASK OBJECT1 SET YOUR SPEED TO 50) ; this need only be mentioned initially
(ASK OBJECT2 SET YOUR SPEED TO 40) ; or a default could have been used
(ASK OBJECT1 PLAN NEXT GRADUALLY FORWARD 300)
; insert (gradually forward 300) into Object1's list of actions for the next tick
(ASK OBJECT2 PLAN NEXT GRADUALLY FORWARD 400)
(ASK OBJECT2 PLAN AFTER 4 TICKS CHANGE YOUR SPEED TO 60); 4 ticks later change speed
```

At this point nothing has happened on the display screen, only the plans have been associated with the objects. To run the plans and record the state there is a special kind of entity, "movies", that cause ticks to be sent to each object and the screen to be recorded. The sending of the message "gradually forward 300" to Object1 causes the following events

```
(ASK OBJECT1 FORWARD 50) ; goes forward 50 units (its speed)
(ASK OBJECT1 PLAN NEXT GRADUALLY FORWARD 250) ;  plans to do the rest next
```

---

1. This paper is not the appropriate place to fully describe the syntax of Director. The last of the following statements means that the message (plan after 4 ticks change your speed to 60) is sent to object2. Four ticks later object2 will receive the imbedded message, i.e. (change your speed to 60). The imbedded message may be any message that the recipient can respond to.

II. An Example from Celestial Mechanics

Suppose we want to simulate the orbits of planets and space ships. One way to do this is to associate with each physical object another object corresponding to its velocity. The velocity actors have their own state and their position in velocity space relative to (0, 0) represents their direction and magnitude. At each tick each physical object's position is updated by adding it to its velocity. The velocity itself may be updated in a similar manner by the thrust of the ship or by the gravitational pull of other massive objects. The tick mechanism provides a means by which the different physical objects can behave in apparent parallelism. Ticks also simplify the physics by reducing the problem to the computation of the change during a small constant unit of time. In this way the integration needed to compute the position and velocity is approximated implicitly by the program. Turtle geometry further simplifies the mathematics by computing the vector additions in velocity space by moving the turtle instead of using trigonometry explicitly. This representation of a velocity vector by the position of a turtle is similar to the approach presented in [Abelson 1975].

First we define the class of physical objects by describing how to make instances of it, how to update the state of an instance and how to compute the gravitational pull caused by an object. A subclass of physical objects, space ships, are defined to do all that physical objects do and, in addition, know how to thrust forward. Suns and planets are subclasses of physical objects with no special behavior. Finally we define the gravitational field which is capable of changing the velocity of any object by exerting the pulls of all the masses that it knows about.

```
(define physical-object object
 ;; make physical-object as a kind of object and send it the following messages
 (set your mass to 10) ;; the default mass
 (receive (make ?instance) now do ;; this enables me to extend the normal behavior
          (ask :self make ,instance) ;; create the object as normal
          (ask ,instance plan next repeat forever update your state)
          ;; on every tick send yourself the message (update your state)
          (ask velocity make (velocity-of ,instance)) ;; make a velocity for object
          instance) ;; return the newly created instance
 (receive (update your state) ;; when I get a message asking me to update my state
         (cond ((ask :self recall your offspring)) ;; if a class do nothing
               (t (ask :self change your position to ;; I update my position by
                       ;; by adding to my current position to the position of my velocity
                       ,(position-sum (ask :self recall your position)
                                      (ask (velocity-of ,:self) recall your position)))
                  ;; I ask the gravitational field at my location to change my velocity
                  (ask gravitational-field
                       apply gravitational forces at
                       ,(ask :self recall your position) to (velocity-of ,:self)))))
 (receive (yield pull at ?place)
          ;; to determine the gravitational pull at the place (G=1 in our units)
          (quotient (ask :self recall your mass) ;; take my mass
                    (square (ask :self yield distance to ,place))
                    ;; divide by the square of my distance to the place to get force per second
                    :frames-per-second ;; divide by this to get force per frame
                    :ticks-per-frame))) ;; divide to get force per tick
```

```
(define gravitational-field something  ;; make the field and send it the following messages
 (receive (apply gravitational forces at ?place to ?velocity)
          ;; for me to apply the gravitational forces at a place to a velocity
          (ask :self exert pulls of ;; I exert the pulls of the masses not at the place
                     ,(remove-any-at-place (ask :self recall your masses) place)
                     on ,velocity at ,place)) ;; on the velocity
  (receive (exert pulls of (?first-mass %rest-of-the-masses) on ?velocity at ?place)
          ;; to exert the gravitational pull at a point of some masses on a velocity
          (ask ,velocity
               move ,(ask ,first-mass yield pull at ,place)
               in direction from ,place to ,(ask ,first-mass recall your position))
          ;; move towards the mass from the place by the pull (acceleration) at that place
          (ask :self exert pulls of ,rest-of-the-masses on ,velocity at ,place))
          ;; and let the rest of the masses exert themselves on the velocity
  (receive (exert pulls of () on ? at ?);; when there are no more masses do nothing
          nil))

(define velocity object);; a velocity is an object so that is can more in velocity space

(define ship physical-object ;; now to define ships
 (receive (thrust forward ?amount) ;; When I'm asked to thrust forward
          (ask (velocity-of ,:self)
               change your heading to ,(ask :self recall your heading))
          ;; I set the heading of my velocity to my own heading
          (ask (velocity-of ,:self) ;; and change my velocity by
               ;; having it go forward the quotient of the thrust and my mass
               forward ,(quotient amount (ask :self recall your mass))))
 (draw using draw-rocket of size))
 ;; and I am drawn by the Draw-rocket procedure applied to my size

(define sun physical-object ;; a sun is also a physical-object
 (set your angle to 10) ;; near enough to a circle (really a 36-agon)
 (set your mass to 100) ;; the default mass of a sun is 100
 (draw using draw-poly of size angle)) ;; I am drawn using Draw-poly of my size and angle

(define enterprise ship  ;; make a ship called the enterprise
   (set your state to (-700 200 45)) ;; put me at any interesting starting state
   (show) ;; show yourself
   (plan next repeat 10 thrust forward 100));; turn on thrusters for the next 10 ticks
```

```
(define sun1 sun  ;; make sun1
   (ask (velocity-of sun1) to back 25) ;; start me off with a velocity of 25 downwards
   (set your size to 100) ;; give it a size
   (set your mass to 7000000)) ;; and a big mass

(define sun2 sun  ;; this one is a little smaller and less massive
   (ask (velocity-of sun2) to forward 80)
   (set your state to (600 0 0)) ;; start off way to the right
   (set your size to 60)
   (set your mass to 3000000))
```

(ask gravitational-field set your masses to (sun1 sun2 enterprise)) ; *tell the field about the objects*

; *Everything is ready to go, so to test it we make a 10 tick movie. It can be seen in Figure 1.*

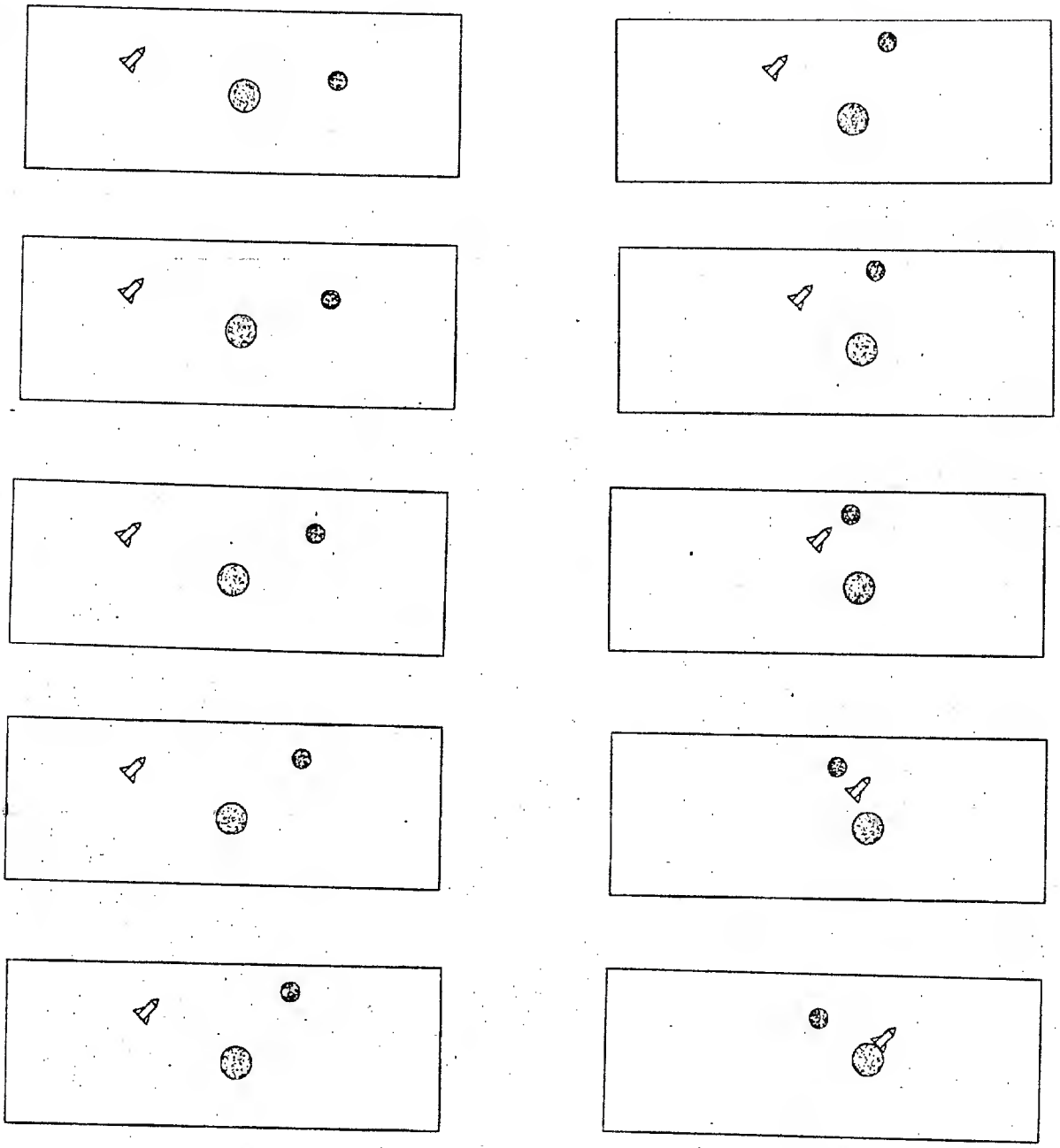
```
(define test-movie-1 movie
 (film the next 10 ticks);; finally make the movie
 (project));; show the movie at default speed and order
```

The advantages of programming in this fashion are many. Computational entities correspond very closely with physically intuitive entities. Corresponding to each object in space there is an object in the program complete with state and a behavioral specification. The gravitational field is also a separate entity which upon request applies the gravitational pull of each mass to any velocity. The ticks reduce the computation to that of calculating the change during a small amount of time. Also the mathematics in the example is kept simple enough for a ten year old by keeping the trigonometry inside of the turtle primitives for moving forward and turning right. It should be clear that the program is very general, that any number of objects can exist and new ones can even be added or old ones removed at any time. Also the accuracy with which the calculations take place are easily controlled by the variable for the number of ticks per frame.[1]

---

1. The time needed to compute many ticks for each frame might be very high though.

Figure 1 --- A Simple Test Run with Two Suns and a Ship



III. Efficiency and the Distribution of Control and Data

Control and data are distributed in the previous examples of the use of ticks and objects. There is little doubt that this reduces the conceptual complexity of the programming but it poses many questions regarding the efficiency of programming graphics in this manner. For example, the lack of any global agenda or schedule might lead one to suspect that the distribution of the planned actions and their times of occurrence is less efficient. The argument goes as follows. If an object plans to do some action many ticks from now and nothing until then, then if control was based upon a global agenda then nothing need happen until that time. With the information in the agenda spread out in the objects involved the object with something to do much later still must be sent ticks in order to decrement the time it plans to do the action. This seems physically intuitive, but needlessly inefficient. The cost of an actor processing a tick, however, can be very small. Moreover, the distribution of the plans makes the changing of plans much simplier. An object can take its plans and modify them and there is no global structure that also needs to be updated.

Planning with a tick mechanism is not restricted to plans with respect to a particular time. To plan an action to happen when a particular event happens or condition is met can be done two ways. Either the actor involved can ask other actors to inform it of some event or upon every tick it determine if some condition is true. In this way an object can plan to explode when it collides with another or to go forward when some other actor has finished going forward.

Suppose we want the ships to melt upon collision with a sun and explode if colliding with anything else. Then using ticks and messages we can arrange that each ship asks the other objects where they are on every tick, determines whether they are colliding and behaves accordingly. An alternative convention is to arrange that on every ticks an actor corresponding to space (or several actors representing regions in space) checks for collisions of objects within it. This scheme is less general, but usually more efficient, than the one where each object asks each other for its position. For example, to have the Enterprise explode or melt upon collision ask it the following:

```
(ask enterprise plan to (cond ((ask ,other are you a sun) ;; if the other is a sun
                               '(melt)) ;; then the action is to melt
                              (t '(explode))) ;; otherwise it is to explode
                        ;; only if receiving a message about colliding with some other
                 after receiving colliding with ?other)
```

If one has a multi-processor system with many processors then a tick mechanism can easily be programmed to take advantage of them. All the events that occur within a tick are unordered except for any requirements to serialize the acts of individual actors. The events are grouped by the object involved and so in terms of locality of data, one can optimize by running those actions of the same object on the same processor. The advantages of having ticks are great if one is running on parallel hardware since there is no global data structure that must be kept consistent and easily accessible.

IV. Synchrony Problems

When processes are being run independently one occasionally runs into synchrony problems. The most common occurrence of such problems is when many objects try to do something that only a few can do at once. The simple example of how to handle this within a tick framework that we shall explore is how to define doors such that at most one object can go through a door on a tick. We want a fair solution, so that those waiting the longest for a door get through and no one need wait forever. A solution that we shall present is one where each object asks the door for permission before entering. The object need not wait around in a line, it is more like getting a number at a crowded store. Each door keeps a queue of the objects that want to go through it. A door is defined to inspect its queue at each tick and if it is not empty the door removes the object at the head of its queue and asks the object if it will enter now. If it does not want to enter the door any longer (maybe it went through another door in the meantime) then the next object on the queue is asked if it will go through now and so on.

The programming of this in Director can be accomplished easily as follows:

```
(define door something  ;; define a door as follows
   (receive (place me on your queue ?wanderer);; a wanderer wants to go through me
            (ask (queue-of ,:self) enqueue ,wanderer));; so I put it at the end of my queue
   (receive (grant permission to front of queue)
            (cond ((not (ask (queue-of ,:self) empty?)) ;; only if the queue is not empty
                   (let ((front-of-queue (ask (queue-of ,:self) dequeue)))
                        ;; I take the first one off the queue and call it front-of-queue
                        (cond ((ask ,front-of-queue will you go thru ,:self door now?)
                               ;; if it is willing to go through me
                               (print '(,:self door letting ,front-of-queue thru
                                                 at ,(ask :self recall your time))))
                               ;; then print event for demonstration and testing
                               (t (ask :self grant permission to front of queue)))))))
                                  ;; if the front of the queue changed its mind
                                  ;; then try again with the next in line
```

```
   (receive (make ?a-door) now do ;; when making a new door
            (ask :self make ,a-door) ;; make it as usual but
            (ask queue make (queue-of ,a-door)) ;; also make a queue for the door
            (ask ,a-door plan next repeat forever grant permission to front of queue)
                 ;; every door should plan to always grant permission
                 ;; to the head of its queue
            (ask ,a-door set your time to 0) ;; initialize its time
            (ask ,a-door plan next repeat forever increment your time by 1)))
            ;; the time is only used for testing and demonstration

(define wanderer something ;; define the objects that wander around and go thru doors
   (receive (will you go thru ?door-name door now?)
            ;; when asked if I will go thru a door now
            (cond ((ask :self recall your (wanting-to-go-thru ,door-name))
                   ;; if I recall wanting to go thru that door
                   (ask :self forget your (wanting-to-go-thru ,door-name))
                   ;; then I forget wanting to go thru the door
                   (ask :self go thru ,door-name door)
                   ;; and actually go thru the door
                   t))) ;; and respond true to the question
   (receive (go thru ?door-name door)
            ;; this is where the wanderer would really go thru the door
            nil)
   (receive (want to go thru ?door-name door)
            ;; If I want to go thru a particular door
            (ask :self set your (wanting-to-go-thru ,door-name) to t)
            ;; then I remember that I want to go thru it
            (ask ,door-name place me on your queue ,:self)))
            ;; and ask the door to put me on its queue

; To test this out we create two doors and a few wanderers and start them going.


(define oak door) ;; create an oak door

(define pine door) ;; create a door named pine

(define lazy1 wanderer ;; create a wanderer named
  (want to go thru oak door)) ;; who wants to go thru the oak door

(define lazy2 wanderer ;; create another named lazy2
   (want to go thru pine door)) ;; who wants to go thru the other door
```

```
(define greedy1 wanderer ;; create another named greedy1
   (want to go thru pine door) ;; who wants to go through both doors
   (want to go thru oak door))

(define greedy2 wanderer ;; as does another wanderer named greedy2
   (want to go thru oak door)
   (want to go thru pine door))

(define sensible wanderer ;; sensible is another wanderer
   (receive (go thru ?door-name door) now do ;; who when going thru a door
            (ask :self go thru ,door-name door) ;; does the usual for a door
            (ask :self forget your (wanting-to-go-thru ?)))
            ;; and forgets about any other doors that it might have wanted to go thru
   (want to go thru pine door) ;; wants to go thru either
   (want to go thru oak door))

; To run this we have the default universe send out ticks to those with something to do next,
; in this case Oak and Pine.

(ask default-universe run for 5 ticks) ;; send out ticks to everyone five times

(PINE DOOR LETTING LAZY2 THRU AT TIME 1) ;; these are printed out by each of the doors
(OAK DOOR LETTING LAZY1 THRU AT TIME 1)
(OAK DOOR LETTING SENSIBLE THRU AT TIME 2)
(PINE DOOR LETTING GREEDY1 THRU AT TIME 2)
(PINE DOOR LETTING GREEDY2 THRU AT TIME 3)
(OAK DOOR LETTING GREEDY1 THRU AT TIME 3)
(OAK DOOR LETTING GREEDY2 THRU AT TIME 4)
```

V. Comparisons with Other Parallel Graphics Systems

Several animation systems permit parallelism that is described and controlled via graphical input. The approach taken in this paper is not an alternative to these demonstrative systems but rather is complementary. One alternative approach was taken by Pfister in the system called Dali [Pfister 1974]. Dali is programed by specifying *demons* which fire when their triggering conditions become true. The use of ticks combined with serializers [Atkinson 1978] is both simplier and more general since it does not make any restrictions upon how information can flow.

Some other languages are too similar to what is presented here to form any important comparisons. For example, Smalltalk [Kay 1977] can easily be extended to have ticks. Simula [Birtwistle 1973], a language which strongly influenced both Smalltalk and Director, could also be changed slightly to support quasi-parallelism for graphics. As we have seen, ticks permit the specification of any condition for an event, while a global agenda sorted by time as in Simula restricts one to a temporal specification.

VI. Conclusions and Directions for Future Research

One wants one's programs to reflect the structure of the task. Dynamic graphics involves the display of changes of many different elements and their features. It has been argued that an object-oriented parallel approach reflects this. This approach is also physically intuitive in its stress on locality and modularity. Programming in this style, one can make use of powerful metaphors from physics and think of each entity as a physical object that is affected only by other actors that send it messages and that behaves independently having its own clock. Another very useful metaphor that a programmer can make use of is that of a society. Just as in societies we are familiar with, there are various structures of command and information flow that map over to object-oriented computation.

One direction of future research is to find other powerful computational concepts for the conceptualization of the display of changing images. Turtles, ticks, and objects are both programming language constructs and ways of thinking about one's problems. There needs to be more of them. For example, perhaps the notion of an activity that an object is engaged in should be explicitly represented as an actor. In that way it could receive messages and change its plans in accordance with new events. One might also consider extending the physics metaphor. Perhaps all events should be viewable only from a "frame of reference" in a way analogous to relativity. The generalization of this idea of taking the observer into account should apply to all events, including, of course, the viewing of a three-dimensional object from a viewpoint. This direction for research is also pointed out in [Kay 1977b], [Moore 1973], and [Bobrow 1977].

A related and equally important direction graphics programming should move is towards the inclusion of much more knowledge into the software. The more the system knows about what the entities being displayed are, how they behave and interact the easier it becomes to use it. The graphics programming has been at too low a level of detail, we should be moving towards systems that know enough so that a user's primary effort is communicating what he or she wants to happen and not how to do it. Much of the research in the artificial intelligence community on

"knowledge-based programming" is very relevant to the task of making images and manipulating them in a convenient manner.

The application of artificial intelligence techniques to computer graphics is called for. One of the authors of this paper is engaged in creating a system capable of producing simple non-representational narrative cartoons in response to a vague, incomplete, high-level description [Kahn 1977b]. The system knows enough about how characters should move and look in order to establish a personality, convey an emotional state, or an interpersonal interaction. Animation is more than the simulation of a world, its production entails inferences, heuristics, and knowledge.

Acknowledgements

We wish to thank Henry Lieberman, Andy diSessa, Bill Kornfeld and Gerry Sussman for their very helpful criticism of earlier drafts of this paper. Henry Lieberman, Danny Hillis, Seymour Papert, and the work of the Learning Research Group at Xerox Parc were a source of many of our ideas. The support of the MIT Artificial Intelligence Lab was crucial. One of the authors (Kahn) is very grateful to IBM for providing a fellowship that gave him the time to explore this and other topics.

VII. Bibliography

[Abelson 1975] Abelson, H., DiSessa A., Rudolph L.
"Velocity Space and the Geometry of Planetary Orbits," American Journal of Physics, July 1975

[Baker 1977] Baker, H. and Hewitt, C.
"The Incremental Garbage Collection of Processes" SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, August 15-16, 1977 University of Rochester

[Birtwistle 1973] Birtwistle, G., Dahl O., Myhrhaug B, and Nygaard K.
Simula Begin, Auerbach Publishers, Inc., Philadelphia, Parallel. 1973

[Goldberg 1976] Goldberg, A. and Kay, A. editors
"Smalltalk-72 Instruction Manual"
The Learning Research Group, Xerox Palo Alto Research Center, March 1976

[Goldstein 1974] Goldstein, I. P.,
Understanding Simple Picture Programs, MIT AI Laboratory AI-TR-294, September 1974

[Goldstein 1975] Goldstein I., Lieberman H., Bochner H., Miller M.
"LLOGO: An Implementation of LOGO in LISP" MIT-AI Memo 307, March 4, 1975

[Hewitt 1975] Hewitt C., Smith B.
"Towards a Programming Apprentice" IEEE Transactions on Software Engineering SE-1, March 1975

[Hewitt 1977] Hewitt, C. and Atkinson, R.
"Parallelism and Synchronization in Actor Systems"
Record of 1977 Conference on Principles of Programming Languages, Jan. 17-19, 1977, 267-280

[Kahn 1976] Kahn, K.
"An Actor-Based Computer Animation Language", Proceedings of the SIGGRAPH/ACM Workshop on User-Oriented Design of Computer Graphics Systems, Pittsburgh, Pa., October 1976

[Kahn 1977a] Kahn, K.
"Three Interactions between AI and Education",
Machine Intelligence 8 Machine Representations of Knowledge,
eds. Elcock E. and Michie, D., Ellis Horwood Ltd. and John Wylie & Sons, 1977

[Kahn 1977b] Kahn, K.
"A Computational Theory of Animation", Massachusetts Institute of Technology,
AI Working Paper #145, April 1977

[Kahn 1977c] Kahn, K., Lieberman H.
"Computer Animation: Snow White's Dream Machine",
Technology Review, Vol. 80, No. 1, October/November 1977, pp. 34-46

[Kahn 1978] Kahn, K.
"Director Users Guide", Forthcoming Massachusetts Institute of Technology, AI Memo, 1978

[Kay 1977a] Kay, A., Goldberg A.
"Personal Dynamic Media", Computer, IEEE, March 1977, v. 10, n. 3, pp 31-41

[Kay 1977b] Kay, A.
"Microelectronics and the Personal Computer", Scientific American, September 1977

[Newman 1971] Newman, W.
"Display Procedures", CACM, Vol. 14, No. 10, Oct. 1971

[Palme 1977] Palme, J.
"Moving Pictures Show Simulation to User",
FOA Rapport, Swedish National Defense Research Institute, April 1977

[Papert 1971a] Papert S.
"Teaching Children Thinking", MIT-AI Memo 247, October 1971

[Papert 1971b] Papert S.
"Teaching Children To Be Mathematicians vs. Teaching About Mathematics",
MIT-AI Memo 249, July 1971

[Pfister 1974] Pfister, G.
The Computer Control of Changing Pictures,
MIT Project MAC Technical Report TR-135, Project Mac, 1974